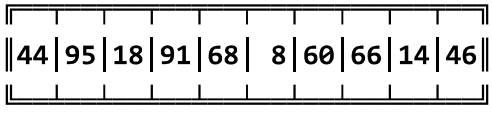
Быстрые сортировки: QuickSort и MergeSort.

Quick-подход

Суть этой методики - рекурсивно разделять объект обработки на меньшие части вплоть до терминального случая. Под словом “меньшие” здесь понимается произвольное уменьшение задачи - может быть уменьшение объёма обрабатываемых данных, может быть уменьшение допустимой области измненения данных, может быть ещё что-то. Собственно, речь может идти и об определённом увеличении каких-то характеристик. В коне счёте речь идёт о решении подзадач рекурсивно, т.е. под уменьшением подразумевается приближение к терминальному случаю рекурсии. Да, подзадачи разбиваем на подзадачи и т.д. вплоть до терминального случая.

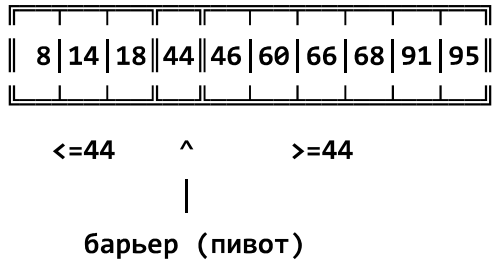
QuickSort

Объект задачи - массив данных, который надо отсортировать. Мы будем сортировать просто int’ы, как обычно, но понятное дело, что таким же образом можно сортировать что угодно, лишь бы оно было сортируемо. Разделение задачи на подзадачи - главный вопрос метода - исполняется совершенно естественно: массив делится на 2 части естественным образом вокруг некоторого барьера - элементы массива переставляются так, чтобы сначала шли элементы, не превосходящие величины барьера, а затем - элементы, не меньшие барьера. Элементы, чья величина равна величине барьера, могут распределяться совершенно произвольно, они даже не обязаны идти подряд. В результате получаем две части, элементам которых не нужно выходить за рамки своей части. Это означает, что после того, как мы (рекурсивно) отсортируем полученные части, делать вообще больше ничего не надо - именно это было целью предобработки, именно так она организована. Давайте в качестве барьера возьмём первый элемент массива. Посмотрим на пример



Разделим массив первым элементом (44) так, чтобы все числа, меньшие 44, располагались

левее от него, а все числа, бОльшие 44, располагались справа от него:



Барьер, который обычно называют пивотом, в результате становится на своё место, перемещать его больше не надо. Если теперь по отдельности отсортировать два подмассива - от нулевого до второго элемента и от четвёртого до последнего, девятого, то в итоге мы получим отсортированный массив.

Рекурсивная схема алгоритма, кажется, уже совершенно ясна. И тут самое время поговорить об окончании рекурсии, о приходе любых её путей к терминальным случаям. Что же нужно для окончания рекурсии? Понятно, что терминальный случай здесь – массив из одного элемента. И совершенно очевидно, что мы с каждым шагом приближаемся к терминальному случаю. Ведь один элемент в результате разделения становится на своё место и выпадает из дальнейшей обработки. Так что даже если одна из двух частей, подлежащих рекурсивной обработке, окажется пустой (вполне возможный вариант, если пивотом оказался наименьший или наибольший элемент массива), то другая часть массива всё равно будет короче всего массива хоть на один элемент.

Тут можно немного поговорить о сложности сортировки, что хотя в худшем случае она окажется $O(N^2)$, но в среднем алгоритм работает существенно быстрее. Говоря точнее, его сложность в среднем окажется $O(N \cdot \log_2 N)$ или просто $O(N \cdot \log N)$. Обсуждать доказательство этого мы точно не будем. Упомянуть можно, но, кажется, лучше даже не упоминать здесь какие-то характеристики такого плана. Достаточно заметить, что за счёт простоты реализации, QuickSort работает быстрее прочих быстых алгоритмов (MergeSort, о котором погворим сегодня, и HeapSort, о котором поговорим в следующий раз). Но, повторю, кажется, что здесь не стОит заморачиваться на анализ скорости/сложности алгоритма, достаточно убедиться в его быстродействии эмпирически.

Общая схема метода представляется вот таким почти кодом:

```
func (l Collection) QSort() {
    if len(l) <= 1 { return } // терминальный случай
    // === Разделение. ===
    // В результате пивот располагается на позиции #division,
```

```
    // слева от неё все элементы не больше пивота,
    // все элементы справа не меньше пивота
    // === Рекурсивные вызовы. ===
    l[:division].QSort()
    l[division+1:].QSort()
}
```

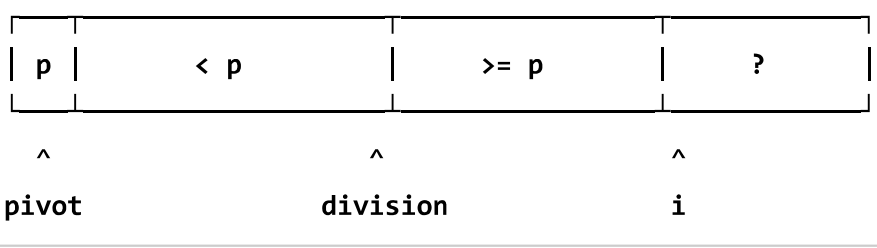
Остаётся самое трудоёмкое в методе - разделение. Хотя называть это действие трудоёмким звучит слишком громко. Тем более, что что-то подобное уже делали во втором-третьем семестрах. Собственно, в третьем семестре этот разговор звучал явно - разговор об инвариантах. Идея понятная - придумать подходящий инвариант цикла и применить его. Очень хочется, чтобы он был

1. понятным и легко реализуемым
2. приводил к эффективной реализации

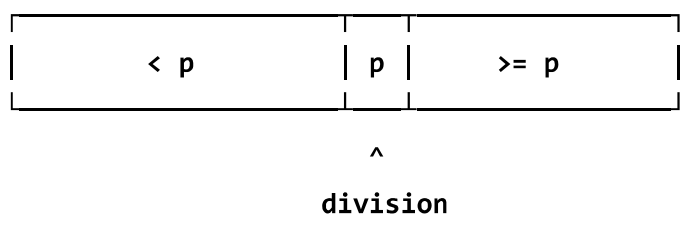
Вариантов тут можно придумать много. Начнём с очень простой, понятной и действительно легко реализуемой схемы разделения.

Разделение-1: двигаемся по массиву в одном направлении (слева направо).

Схема инварианта цикла понятна из рисунка:



В начале элемент, стоящий на 0-ом месте считаем пивотом, а индекс i устанавливаем равным 1. Двигаем индекс i слева направо, уменьшая постепенно область неопределённости до конца и поддерживая, конечно, инвариант. В конце, когда область неопределённости станет пустой, поменяем местами пивот с элементом на позиции division и получим то, что надо:

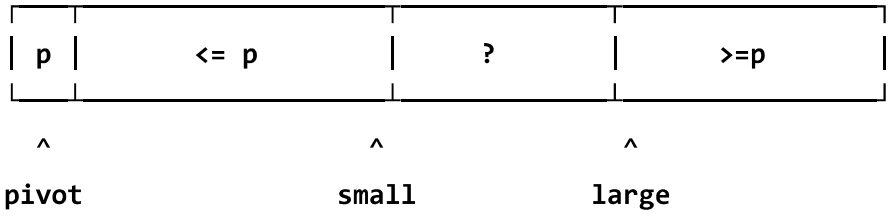


Реализация этой схемы разделения вот:

```
func (l Collection) QSort() {
    if len(l) <= 1 { return }
    // Разделение.
    //    Инициализация.
    pivot := l[0]
    division:= 0
    //    Цикл.
    for i, x := range(l) {
        if x < pivot {
            division++
            l[i], l[division] = l[division], l[i]
        }
    }
    //   Завершение разделения.
    l[0], l[division] = l[division], l[0]
    // Рекурсивные вызовы.
    l[:division].QSort()
    l[division+1:]. QSort()
}
```

Разделение-2: двигаемся по массиву с обоих краёв навстречу друг другу.

Схема инварианта цикла показана из рисунке:



Как и в одностороннем варианте постепенно уменьшаем область неопределённости, поддерживая инвариант. Только двигаемся с двух сторон - и слева, и справа. В конце поменяем местами пивот с элементом на позиции division и получим то, что надо. Подробности - ниже, а начнём с того, что сразу посмотрим на реализацию этой схемы:

```
01.    func (l Collection) QSort() {
```

```
02.         if len(l) <= 1 { return }
03.
04.         pivot := l[0]
05.         small, large := 1, len(l)-1
06.         for {
07.             for small < len(l) && l[small] < pivot { small++ }
08.             for l[large] > pivot { large-- }
09.             if small >= large { break }
10.             l[small], l[large] = l[large], l[small]
11.             small++
12.             large--
13.         }
14.         l[0], l[large] = l[large], l[0]
15.         l[:large].QSort()
16.         l[large+1:]. QSort()
17.     }
```

А теперь мы её обсудим.

Инициализация процесса:

- пивот, как и в первой схеме, находится на 0-ой позиции, индекс small указывает на первый элемент, индекс large - на последний.

Цикл:

- двигаем левый индекс (small) слева направо, пропуская значения, меньшие pivot, оставляем их на своих местах; при этом мы предотвращаем выход указателя за правый край массива (а вдруг, например, все элементы массива меньше выбранного пивота);
- двигаем правый индекс (large) справа налево, пропуская значения, большие pivot, оставляем их на своих местах; при этом беспокоиться о выходе за левый край массива не надо - элемент на 0-ом месте автоматически остановит указатель large;
- после того, как оба указателя остановились возможны следующие случаи:
    - small < large. Меняем местами соответствующие элементы и продвигаем указатели на одну позицию (каждый в свою сторону) - в результате инвариант цикла сохраняется, а указатели не выходят за границы массива. Почему не выходят? До увеличения на 1 выполнялось условие small < large, значит после увеличения small на 1 будет выполняться условие small <= large, а large не выходит за границу массива. До уменьшения на 1 выполнялось условие large > small, а small > 0, значит после уменьшения на 1 выполняется условие large > 0. Это гарантирует отсутствие неприятностей при следующем прохождении цикла в строках 06-13.

- small = large. Это возможно только в том случае, если они указывают на значение = пивоту. Выходим из цикла и выполняем один, не нужный в данном случае, обмен в строке 14 - меняем местами два одинаковых значения, но проще и быстрее выполнить этот обмен, чем ставить лишнюю проверку. В результате large (и small, но это неважно) указывает на знаяение = pivot, слева от него все числа <= pivot, а все значния справа >= pivot
- small > large. Кажется, что такое невозможно - ведь если такая ситуация сложилась, то, вроде бы, предыдущим действием мы уменьшили large в цкиле в строке 8. Значит перед этим действием было large = small. small смотрел на значение >= pivot (условие выхода из цикла в строке 07) и large должен был на нём остановиться. Но более тщательный взгляд показывает, что такая ситуация возможна, а именно: при предыдущем прохождении цикла 06-13 в строке 09 выполнялось условие `small + 1 = large` ; после выполнения строк 10-12 small и large обменяются своими значениями, окажется `small - 1 = large` и мы зайдём в цикл 06-13. При этом small будет указывать на значение >= pivot, а large - на значение <= pivot, и циклы в строках 07 и 08 выполняться не будут. А строка 09 выбросит нас из цикла 06-13.

Красиво, правда? А красота - это фактор важный. Очень часто красивое решение оказывается более эффективным. Не будем влезать в неведомые философские чащи и выяснять, что же это такое - красивый алгоритм и красивый код, просто порадуемся. Да, и такая схема разделения действительно более эффективна. Хотя ошибиться при написании кода в двусторонней схеме гораздо проще.

Рандомизация выбора пивота.

Если исходный массив хорошо перемешан, то каждый раз он разделяется примерно на равные части - неважно, где и как выбирать пивот. Так что мы поступаем самым простым способом - берём пивот в нулевом элементе, это чуть-чуть, но упрощает и чуть-чуть, но ускоряет реализацию. Но что будет, если массив уже почти отсортирован? А такая ситуация встречается нередко. Давайте для простоты будем считать, что начальный массив отсортирован полностью. Тогда мы будем каждый раз отщеплять от массива по одному элементу, одна из частей после разделения окажется пустой, а во вторую войдут все элемент, кроме одного. И эту часть приждётся снова всю прсматривать и т.д. В результате мы не только получим очень глубокую рекурсию (что чревато переполнением стека вызовов), но и, самое главное, дикое торможение. Бенчмаркинг показывает это с кристалльной ясностью, так что заниматься более точным анализом мы не станем. Спасает от таких эффектов случайный выбор пивота. Изменения минимальные, а эффект в случае почти отсортированных массивов - налицо. Вариант реализован в **программе qsort2_RandomPivot.go**. Двойка в названии означает, что для используется схема разделения с движением по массиву с двух сторон.

Вариант Сэджвика.

Мы уже видели, что вызов функции - операция недешёвая. А в наших программах громадная часть рекурсивных вызовов приходится на сортировку очень маленьких массивов. Элементы массивов, появляющихся пр рекурсивных вызовах, при сортировке не выходят за рамки этих массивов. Так может быть есть смысл не сортировать маленькие массивы QuickSort'ом, а применить к ним какой-нибудь простенький алгоритм, например, сортировать их простыми вставками или ещё как-нибудь. Сделать это легко: если длина массива достаточно мала, то сразу переводить обработку в терминальный случай, а обработка терминального случая будет состоять в исполнении какой-нибудь простой сортировки.

Однако Роберт Сэджвик (Robert Sedgewick) предложил гораздо более элегантное решение: не надо сразу сортировать короткие массивы. Ведь элементы в них и так никуда не денутся - в окончательном порядке они смогут переместиться только в пределах этого массива, т.е. совсем недалеко, массивы-то маленькие. Поэтому просто оставляем короткие массивы в впокое при рекурсивных обходах, а потом, после окончания рекурсии, пройдёмся один раз по всему массиву сортировкой простыми вставками. Поскольку каждый элемент может сдвинуться очень недалеко, то эта сортировка будет работать быстро. И опыт показывает, что такая политика себя оправдывает - действительно такоая модификация QuickSort даёт ощутимый выигрыш в скорости.

"Разделяй и властвуй"

Называют также этот подход divide et impera (это на латыни) или divide and conquer (это по-английски). Но не в этом суть. А суть подхода, как и в Quick-подходе, состоит в следующем: разбиваем решаемую задачу на две или несколько таких же подзадач, но меньшего размера. Скорее всего после получения решения подзадач требуется как-то комбинировать решения подзадач для получения ответа к исходной задаче.

Декомпозиция здесь носит более насильственный характер - мы сразу определяем размер частей, на которые будем делить массив данных. А делить мы его будем на две (примерно) равные по размеру части, сортируем эти части рекурсивно, а затем склеиваем полученные отсортированные части в один отсортированный массив.

Иначе говоря, если в QuickSort'е мы сначала как-то переставляем элементы так, что нам оставалось отсортировать полученные части полностью независимо друг от друга, что мы и делали рекурсивно, то в MergeSort мы сначала сортируем две половины массива, а потом

как-то комбинируем полученные результаты. То есть, в отличие от Quick-подхода, здесь всё наоборот – сначала просто, бесхитростно так, разделяем массив на две примерно равные по размеру части, безо всякой предобработки, затем рекурсивно сортируем полученные части, а уже потом как-то переставляем элементы.

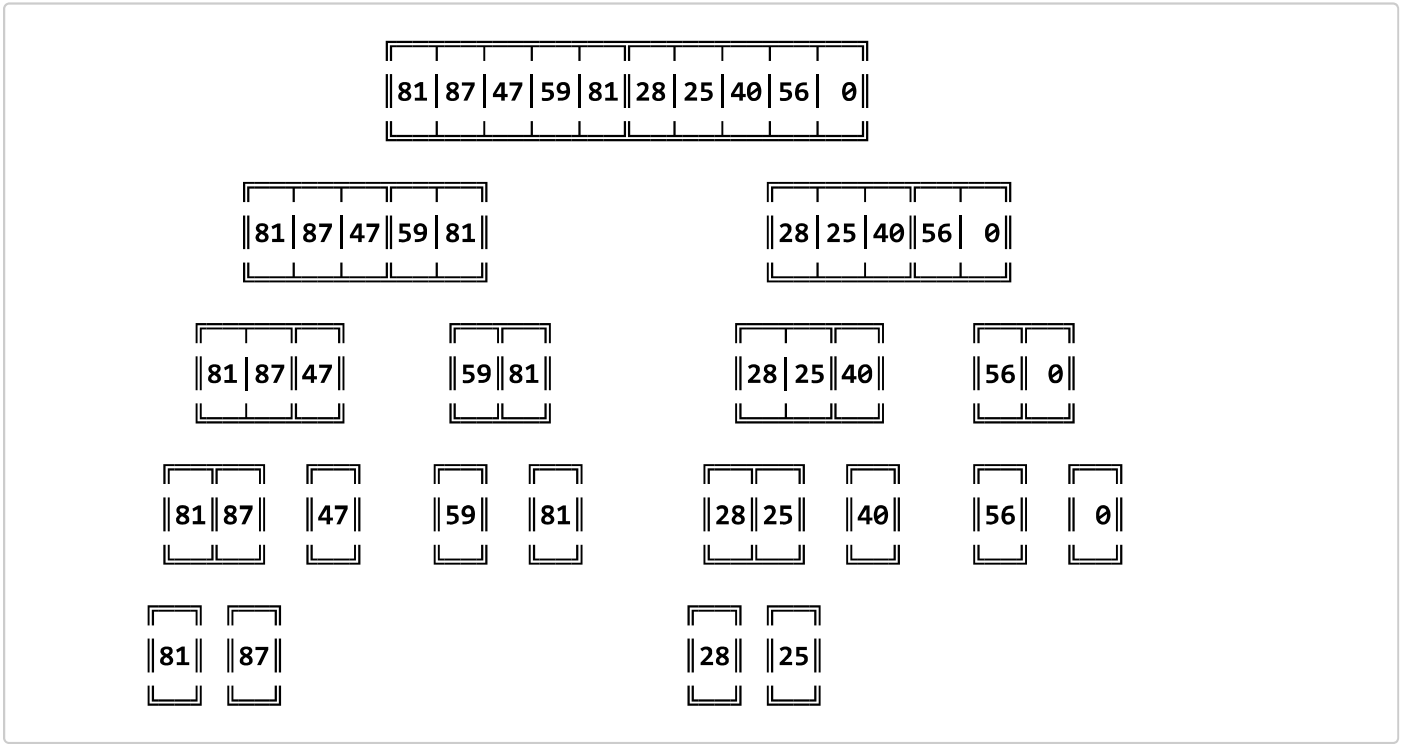
Рекурсия опять просто прёт из описания самого подхода. Имеется много классических примеров успешного применения этого подхода. Но мы остановимся сегодня только на одном - на сортировке слиянием - MergeSort.

MergeSort (точнее BinaryMergeSort) - сортировка слиянием.

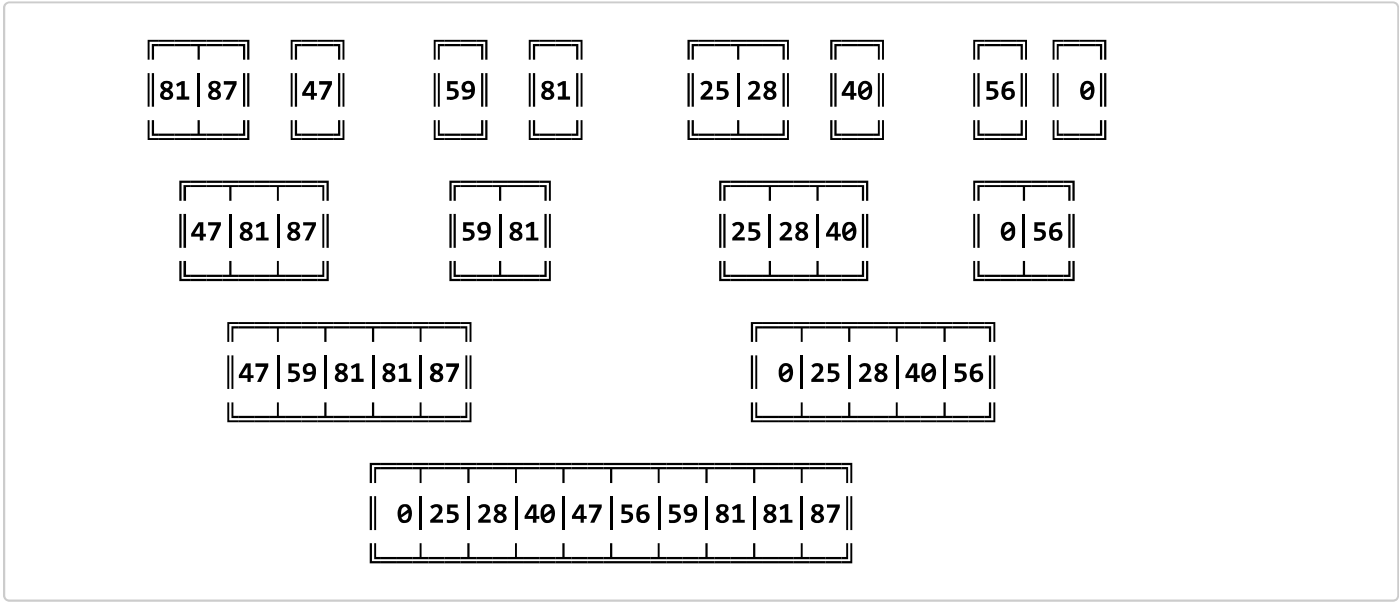
Разбиваем сортируемый массив на две равные части, каждую часть сортируем, а затем отсортированные части сливаем в одну отсортированную. Как и написано выше, эта процедура применяется к каждой из частей пока сортируемая часть массива содержит хотя бы два элемента - чтобы можно было её разбить на две части. Терминальный случай здесь - конечно, одноэлементный массив. Он уже отсортирован - ничего делать не надо.

Проиллюстрируем сортировку слиянием следующей схемой:

Расщепление - прямой ход рекурсии.



Слияние - возвращение из рекурсии.



Более подробно расписывать здесь BinaryMergeSort нет особого смысла - про него написано очень много где. Конечно, когда очень много где, то подавляюще бОльшая часть написанного неудобоварима, мягко говоря. Но есть и очень хорошие описания. Только их, как водится, трудно найти в той куче. Так что эту работу всё-таки проделаем. В приложении А содержится обширный - на 10 страниц - фрагмент из книги *Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: Построение и анализ. 2-е издание. Вильямс, 2005 г.*. Книга классическая, так что у неё есть традиционное сокращённое название - KLRS. Само описание метода в KLRS – очень неплохое. Так что детям этот текст надо дать обязательно - пусть читают. Первую половину (стр.72-77) обязательно. Там несложно, хотя код написан на псевдокоде, основанном на хранении данных в массивах, а мы будем использовать слайсы, что сильно упрощает код и делает его яснее. Но об этом ниже. А вот вторая половина, начиная со страницы 78 - это уже мелкий шрифт, могут и пропустить, там проводится довольно аккуратный анализ сложности (эффективности) метода. Там математика. Небольшая, но детей может и вытошнить.

Значительная часть текста в KLRS посвящена детальному разбору процедуры слияния. Конечно, какую-то реализацию процедуры слияния двух отсортированных массивов в один, тоже отсортированный, мы приведём в примере. Но было бы хорошо, чтобы дети сами написали какой-нибудь свой вариант, а то и два-три варианта. Это **очень неплохое задание на практику – написать реализацию процедуры слияния**. Тем более, весьма вероятно, что что-то подобное дети делали во втором-третьем семестре. А мы на лекции большее внимание уделим собственно декомпозиции вообще. Которая прекрасно ложится на рекурсивную музыку - сущность её, повторю, очень рекурсивна. Ну, и поговорим об особенностях реализации, основанной именно на хранении данных в слайсах. Для этого мы и приведём весь код полностью. Код-то мы им дадим, но пусть попробуют написать слияние сами.

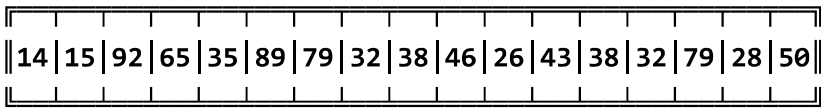
Сортировка слиянием очень легко разрекурсивливается, её совсем легко направить в противоположную сторону - сортировать не от большего к меньшему, что характерно для рекурсивных процедур, а от меньшего к большему, что естественно для циклической организации обработки. В результате мы получим ещё один класс сортировок -

Сортировки серий

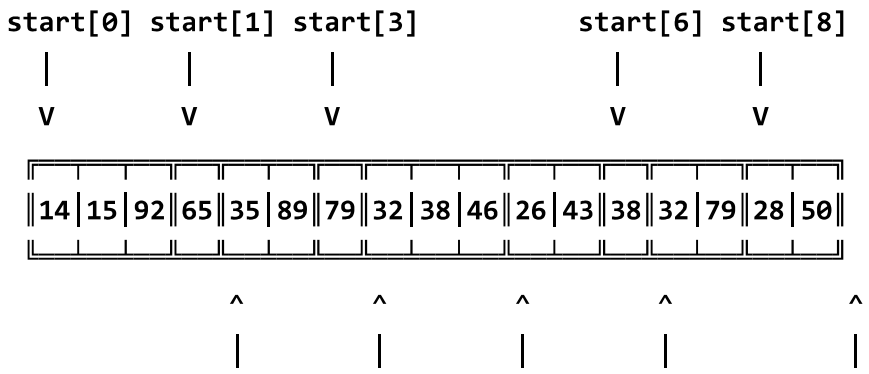
Сериями или возрастающими сериями называем возрастающие (точнее, неубывающие) отрезки массива (или файла, в общем, последовательности данных). По-английски это называется runs (или ascending runs). Суть сортировок сериями основана как раз на процедуре склеивания (Merge) и состоит в том, что мы склеиваем имеющиеся серии в новые более длинные серии, и так до тех пор, пока не останется только одна серия. В рамки этого подхода (парадигмы, если хотите) вписывается громадное количество вариантов, и это действительно очень большая история, но мы только слегка прикоснёмся к этому подходу. Мы будем склеивать серии попарно, для этого у нас уже как раз есть функция Merge, склеивающая две серии, ею и будем пользоваться.

NaturalMergeSort - сортировка естественным слиянием.

В начале серии - это те серии, которые есть в массиве с самого начала.
Вот наш исходный массив:



Первым делом мы проходим один раз по всему массиву и разделяем его на серии.



```
                  start[2]   start[4] start[5] start[7]    start[9]
```

В результате строим массив start, в который помещаем начала всех серий, имевшихся в массиве с самого начала. Начало каждой серии, кроме самой первой (нулевой, точнее говоря) есть одновременно конец предыдущей серии. Вспомним, что слайс [a:b] содержит элементы массива с элемента #a до элемента #(b-1) включительно. Для единообразия включаем в массив start конец последней серии.

Склеиваем нулевую серию с первой, вторую - с третьей, четвёртую - с пятой и т.д. Если серий нечётное количество, то последнюю серию оставляем без изменений.

```
start[0]    start[1] start[2]       start[3] start[4] start[5]
    |           |        |             |        |     |
    V           V        V             V        V     V
   ╔══╤══╤══╤══╦══╤══╤══╦══╤══╤══╤══╤══╦══╤══╤══╦══╤══╗
   ╟14╢15╢65╢92╫35╢79╢89╫26╢32╢38╢43╢46╫32╢38╢79╫28╢50╢
   ╚══╧══╧══╧══╩══╧══╧══╩══╧══╧══╧══╧══╩══╧══╧══╩══╧══╝
```

В примере на рисунке 9 серйи перешли в пять серий. Последняя серия осталась без изменений.

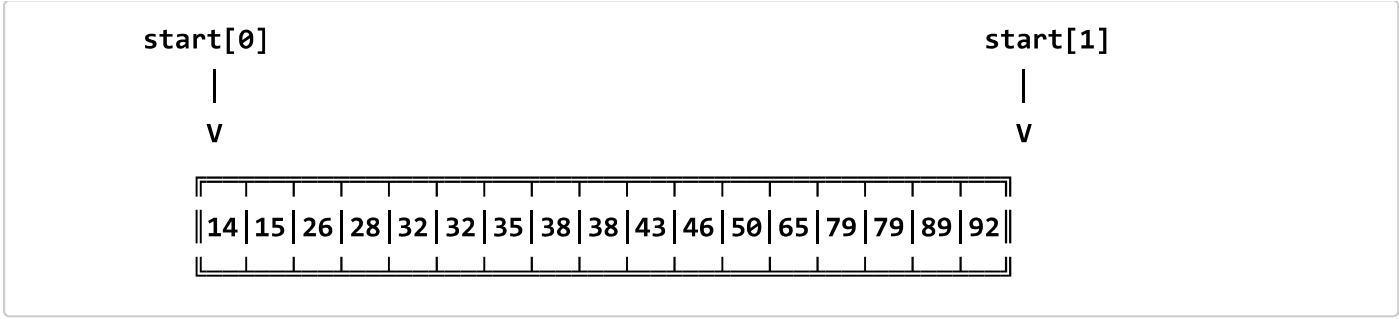
Повторяем процесс, в результате остаётся три серии:

```
start[0]             start[1]                start[2] start[3]
    |                   |                      |     |
    V                   V                      V     V
   ╔══╤══╤══╤══╤══╤══╤══╦══╤══╤══╤══╤══╤══╤══╤══╦══╤══╗
   ╟14╢15╢35╢65╢79╢89╢92╫26╢32╢32╢38╢38╢43╢46╢79╫28╢50╢
   ╚══╧══╧══╧══╧══╧══╧══╩══╧══╧══╧══╧══╧══╧══╧══╩══╧══╝
```

затем две:

```
start[0]                                        start[1] start[2]
    |                                              |     |
    V                                              V     V
   ╔══╤══╤══╤══╤══╤══╤══╤══╤══╤══╤══╤══╤══╤══╤══╦══╤══╗
   ╟14╢15╢26╢32╢32╢35╢38╢38╢43╢46╢65╢79╢79╢89╢92╫28╢50╢
   ╚══╧══╧══╧══╧══╧══╧══╧══╧══╧══╧══╧══╧══╧══╧══╩══╧══╝
```

и, наконец, одна:



Склеиваем серии с помощью уже имеющейся у нас функции Merge. Правила преобразования слайса start тоже вполне очевидны:

- start[2] переходит в start[1], start[4] переходит в start[2], start[6] переходит в start[3] и т.д.;
- если сначала было нечётное количество серий, т.е. если len(start) есть чётное число, то в конец изменённого слайса start дописываем указатель на концовку последней серии - он всегда равен длине сортируемого слайса.

Детали и подробности хорошо видны в коде:

```
func (l Collection) NaturalMergeSort() {
    // инициализация - заполняем слайс start
    start:= []int{0}
    for i:= 1; i < len(l); i++ {
        if l[i] < l[i-1] {
            start = append(start, i)
        }
    }
    start = append(start, len(l))
    // сортировка
    for len(start) > 2 {
        // проходим по всему массиву, склеивая пары соседних серий
        for k:= 0; k < len(start) - 2; k += 2 {
            l[start[k]:start[k+2]].Merge(start[k+1]-start[k])
        }
        // преобразуем слайс start: start[2] -> start[1],
        // start[4] -> start[2], start[6] -> start[3] и т.д.
        k:= 0;
        for {
            k += 2
            if k >= len(start) { break }
            start[k/2] = start[k]
        }
```

```
        start = start[:k/2]
        // если перед этим было нечётное количество серий, то
        // надо добавить конец последней серии - len(l)
        if start[len(start)-1] < len(l) {
            start = append(start, len(l))
        }
    }
}
```

Полный текст находится в **naturalmergesort.go**

И напоследок

Сортировка бинарным слиянием - нерекурсивный вариант

А тут всё совсем просто - та же самая сортировка естественным слиянием, но начинаем с серий длины 1, т.е. каждый элемент представляет из себя серию. В результате после первого прохода по массиву получаем серии длины 2 (кроме, может быть последней), после второго - серии длины 4, потом - серии длины 8, и т.д. Изменения минимальные, приводить здесь код особого смысла нет. Полный текст находится в файле **mergesortunrecursive.go**.

Задачи для самостоятельной работы

1. В тексте про QuickSort с вариантами разбросано несколько потенциальных улучшений алгоритма, некоторые варианты не рассматриваются (например, мы не рассматриваем случайный выбор пивота в односторонней схеме разделения). Хорошо бы поэкспериментировать с вариантами, побенчмарчить их, посмотреть как это влияет на скорость.

- случайный выбор пивота в схеме с односторонним разделением
- выбирать пивот из центра слайса - это спасает в часто встречающемся случае, когда массив почти упорядочен
- сделать случай с массивом из двух элементов терминальным
- попробовать разделять массив на три части, выбирая, соответственно два пивота X и Y; части получаются <=X, от X до Y, >=Y
- вариант с одним пивотом X, но с разделением на три части: X; рекурсивный вызов применяем, разумеется только к крайним частям. Этот вариант хорошо работает в

случае, когда в исходном массиве много повторяющихся элементов
- улучшение Сэджвика в схеме с односторонним разделением
- случайный выбор пивота + улучшение Сэджвика
- во всех вариантах алгоритма со случайным выбором пивота заменить случайный выбор на выбор пивота из середины отрезка; это может ускорить выполнение алгоритма за счёт отказа от обращения к рандом-генератору (а это штука небыстрая, как мы видели), при этом стандартные и притом нередко встречающиеся варианты с почти упорядоченным массивом всё равно будут обрабатываться быстро.
- во всех рекурсивных версиях программы дрбавить выбор порядка выполнения рекурсивных вызовов: сначала сортировать рекурсивно меньший отрезок, а потом больший.
- в варианте Сэджвика не откладывать сортировку простыми вставками, а сразу сортировать простыми вставками короткие отрезки, т.е в терминальных случаях исполнять сортировку простыми вставками или ещё какую-нибудь простую сортировку, например
    - сортировку “выбором наименьшего. Идея этой сортировки простая: выбираем наименьший элемент отрезка и меняем его местами с начальным элементом; в результате на нулевом месте стоит “правильный” элемент и мы повторяем этот процесс для оставшегося отрезка; так заполняем “своими” элементами первое место в отрезке, второе, третье и т.д. до конца.
- вариант Синглтона: в качестве пивота берём среднее (не среднее арифметическое, а именно среднее, медиану) трёх чисел - того, которое стоит в начале отрезка, того, которое стоит в конце отрезка, и того, которое стоит посередине. Если длина отрезка меньше трёх, то обрабатываем непосредственно. Собственно, непосредственная обработка нужна только для отрезков из двух элементов.

А **задание** будет - поэкспериментировать, какие из приёмов помогают, дают рост эффективности, какие - нет, насколько эффективны те или иные приёмы. Как влияют эти приёмы на работу алгоритма с

```
- случайным массивом
- почти отсортированным в правильном порядке массивом (или даже просто с уже отсорти
рованным массивом)
- почти (или полностью) отсортированном в обратном порядке (по убыванию вместо возра
стания) массивом
- возможно, что стОит рассмотреть особо ещё какие-нибудь часто встречающиеся ситуаци
и
```

2. QuickSearch. Обещанное описание алгоритма для поиска K-го по величине числа среди заданного массива чисел (интеллигентно это называется поиском K-й порядковой статистики).

Идея простая и в понимании, и в реализации, так что задачка совсем простенькая

получается. А суть алгоритм состоит в следующем.

Исходная задача: имеется массив чисел (слайс, конечно); требуется найти K-й по величине его элемент. Для простоты будем считать, что не только позиции в массиве нумеруются с 0, но и элементы по величине тоже нумеруются с нуля, т.е. наименьший элемент мы будем называть “нулевой по величине элемент”. Может это и звучит странно, ну так переделать легко.

Алгоритм:

```
- выбираем пивот и выполняем разделение, пивот в результате оказывается на месте номер P
    - если P = K, то пивот и есть искомый элемент
    - если P > K, то остаётся найти K-й по величине элемент в левом отрезке разделения
    - если P < K, то остаётся найти (K-P)-й элемент в правом отрезке, ведь пивот и все элементы, оказавшиеся слева от него меньше того элемента, который мы ищем, и их можно не рассматривать.
```